SUPPORTING SPATIAL INDEX STRUCTURES AS ACCESS METHODS IN A DATABASE SYSTEM

BY

KAUSHIK CHAKRABARTI

B.Tech, Indian Institute of Technology, Kharagpur, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

Acknowledgments

I thank my research advisor, Professor Sharad Mehrotra, for his exceptional guidance during the course of this research. He was always available for discussion and motivation.

I thank my academic advisor, Professor Geneva Belford, for her help and guidance.

I am grateful to my parents and other members of my family in India for their constant support and encouragement.

I also thank the members of our research group with whom I worked closely. Among them, I must particularly mention Michael Ortega and Kriengkrai Porkaew for their help and cooperation.

Finally, I thank all my friends at University of Illinois for their friendship.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DBMS Database Management System

GL Granular Locking

PL Predicate Locking

MGL Multigranularity Locking

KRL Key Range Locking

R-tree Rectangle-tree

BR Bounding Rectangle

GiST Generalized Search Tree

List of Relevant Publications

- **Dynamic Granular Locking Approach to Phantom Protection in R-trees**,

  K. Chakrabarti and S. Mehrotra, *1998 IEEE International Conference on Data Engineering , February 1998.*

  (Online version: http://www.computer.org/conferen/proceed/icde/8289/)

- **Concurrency Control in R-trees**,

  K. Chakrabarti and S. Mehrotra, *1st Annual FedLab Symposium, Advanced Displays & Interactive Displays Consortium , February 1997*

  (Online version: http://www-db.ics.uci.edu/pages/publications/index.shtml#TR-MARS-97-01)

- **Integration of Multidimensional Index Structures as Access Methods into a DBMS**,

  K. Chakrabarti and S. Mehrotra, *ARL Workshop on Databases and OO Technologies, November 12, 1997 (Presentation)*

  (Online version: http://www.ics.uci.edu/~kaushik/research/research.html)

Chapter 1

Introduction

Modern database systems are commonly used to store spatial data. Spatial data is usually accessed based on its spatial location. A common operation on spatial data is a search for all objects in an area, for example to find all countries that have land within 20 miles of a particular point. This kind of spatial search occurs frequently in computed aided design (CAD) and geo-data applications, and therefore it is important to be able to retrieve objects efficiently according to their spatial location.

Spatial location usually involves multiple dimensions (e.g., 2-D or 3-D). An index based on objects' spatial location is desirable, but classical one-dimensional database indexing structures are not appropriate for multidimensional spatial searching. Structures based on exact matching of values, such as hash tables, are not useful because a range search is required. Structures using 1-d ordering of key values, such as B-trees and ISAM indexes, do not work because the search space is multidimensional.

To meet the needs of spatial indexing, several multidimensional data structures (e.g., R-trees [9], grid files [20], hB-trees [5]) have been proposed over the last 15 years. These index structures allow search based on multiple attributes simultaneously. Despite extensive research, almost none of the developed data structures are supported as access methods by existing database management systems (DBMSs). [1] Among the primary reasons is the lack of protocols to provide transactional access to data and to guarantee the consistency of the index structures in the presence of concurrent operations. This thesis addresses the problems resulting from the concurrent access to spatial data

[1]An exception to this is the Postgres System [8] developed at University of California at Berkeley,(and its commercial version the Informix Universal Server (IUS) [11]), which supports R-tree as an access method to optimize retrieval of multidimensional data. However, Postgres requires transactions to lock the entire R-tree thereby disallowing concurrent transactions. The strategy used by IUS, if different from Postgres, is not public.

in the database via spatial access methods.

Concurrent access to data through a multidimensional index structure introduces two independent concurrency control problems. First, techniques must be developed to prevent concurrent insertions, deletions and updates from violating the consistency of the data structure. Usage of the standard *two-phase locking* protocol [6] for this purpose results in the data structure becoming a bottleneck and thus poor performance. Many approaches that exploit the structure and the semantics of the operations to provide high concurrency have been developed for B-trees [6]. These approaches need to be generalized to multidimensional data structures. Initial steps in this direction can be found in [12, 22] which discuss techniques to support concurrent operations on the R-tree and the grid file respectively.

Second, techniques must be developed to protect the ranges specified in the retrieval from subsequent insertions and deletions before the retrieval commits. Such insertions and deletions are referred to as the *phantoms*. Consider a *range query* over a B-tree based on the salary of an employee to retrieve all employees with salary between 10K and 20K. To answer such a query, the B-tree is traversed to retrieve the pointers to the data pages containing the records qualifying the predicate 10K $\leq$ salary $\leq$ 20K. Even if all objects currently in the database that satisfy the predicate are locked, the object-level locks will not prevent subsequent insertions into the search range. These insertions may be a result of insertion of new objects or rolling-back deletions made by other concurrent transactions. So if the searcher repeats its scan, it will find new objects in its range which seem to appear from nowhere (hence referred to as phantoms). The technique used to provide phantom protection in B-trees is *key range locking* (KRL)[15, 17]. KRL, however, relies on the presence of a total order over the underlying data based on their key value and hence is inapplicable to multidimensional index structures.

This thesis concentrates on developing techniques to provide phantom protection to retrievals when the data is accessed via R-trees. The R-tree is one of the most popular spatial index structures as it provides the best tradeoff between performance and implementation complexity [7]. Several variants of R-trees (R+trees [24], R*-trees [3], Greene's R-tree [7]) have been proposed in the literature in order to optimize its performance. R-trees and its variants can be used for indexing both point and spatial data (data with spatial extents) and is the only multidimensional index structure known to have been incorporated as an access method into a commercial data management system [11]. It has also been implemented as a part of the Paradise parallel data management system [4].

Despite the importance of spatial access methods to emerging database applications, and the requirement to address the phantom problem in order to provide transactional access to data using these data structures, surprisingly little research exists on providing phantom protection to retrievals over multidimensional data structures. The only other work we are aware of which is parallel to ours is the approach developed in [13] which uses a modified *predicate locking* mechanism to provide phantom protection over *Generalized Search Trees* (GiSTs) [10]. In contrast, the technique developed in this thesis is based on *dynamic granular locking.* As discussed in [6], although the predicate locking offers potentially higher concurrency, typically the granular locking is *preferred* since the lock overhead of predicate locking is higher compared to that of granular locking. We provide a detailed comparison of the granular locking approach developed in this thesis to the predicate locking approach proposed in [13] in Chapter 4.

The rest of the thesis is developed as follows. Chapter 2 provides the necessary background and discusses the main challenges in developing a granular locking approach to solve the phantom problem in R-trees. In Chapter 3, we present the dynamic granular locking approach to prevent phantoms in R-trees. This is the main contribution of this thesis. We present the performance results in Chapter 4 and compare the granular locking approach to other techniques. Finally, Chapter 5 offers the summary and the concluding remarks.

Chapter 2

Background

To provide the necessary background, in this chapter, we provide an overview of the R-tree data structure [9]. Then, we define the phantom problem and discuss the solutions to the phantom problem in B-trees. Then we discuss why the solution for B-trees cannot be applied to R-trees. We conclude the chapter with a description of the fundamental challenges in developing a granular locking approach to solve the the phantom problem in R-trees.

2.1 Overview of R-trees

An R-tree [9] is a height-balanced tree similar to the B-tree with index records in its leaf nodes containing pointers to data objects. Nodes correspond to disk pages if the index is disk resident, and the structure is designed so that a spatial search requires visiting only a small number of nodes. The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required.

A spatial database consists of a collection of tuples representing spatial objects, and each tuple has a unique identifier which can be used to retrieve it. Leaf nodes in an R-tree contain index record entries of the form

$$\langle I,\ \textit{tuple-identifier} \rangle$$

where *tuple-identifier* refers to a tuple in the database and I is an n-dimensional rectangle which is the smallest possible bounding box of the spatial object indexed. Non-leaf nodes contain entries of the form

$$\langle I,\ \textit{child-pointer} \rangle$$

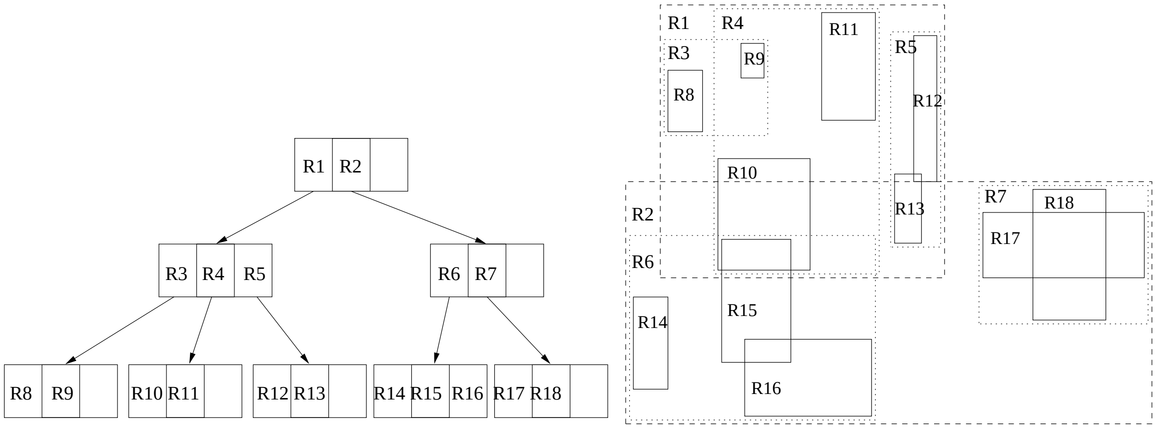

Figure 2.1: An R-tree along with the spatial distribution of the data items.

where *child-pointer* is the address of a lower node in the R-tree and I is the smallest possible rectangle that covers all rectangles in the lower node's entries.

Like B-trees, each node of the R-tree contains between m and M index records unless it is the root where M is the maximum number of entries that will fit in one node and $m \leq \frac{M}{2}$ is a parameter specifying the minfill of a node. All the leaves appear at the same level. Figure 2.1 shows the structure of an R-tree and illustrates the containment and overlapping relationships that can exist between the rectangles. We next describe how the search, insertion and deletion operations are implemented in the R-tree.

- **Search:** Searching in an R-tree proceeds by descending the tree from the root as in B-trees. To search a specific object, the searcher needs to descend only along those paths that may contain the object. Range queries can be either overlap or containment queries. For both types of range queries, search needs to descend along all overlapping paths. Only at the leaf level, the set of objects retrieved differ. While overlap queries retrieve all objects overlapping with the search area, containment queries retrieve only those objects that are completely contained within the search area. Since containment queries are a refinement of overlap queries, locks acquired to protect an overlap query from phantoms also suffice for phantom protection of a containment query. For this reason, we only consider overlap queries in this thesis. It might be possible to devise protocols that acquire less restrictive locks for containment queries providing higher concurrency but that direction of research is not pursued in this thesis.

- **Insert:** To insert an object into the R-tree, the inserter descends along only one path. Algorithm **ChooseLeaf** is invoked to select a leaf node L in which to place the object. The node L whose $L.I$ needs least enlargement to include the object is selected. If the node L was already full, insertion will cause L to split. Algorithm **SplitNode** decides the best possible split. The spitting criterion is to minimize the total area of the two bounding rectangles after the split. The exhaustive algorithm has exponential complexity; so approximate algorithms of quadratic or linear complexity are commonly used. Then the algorithm **AdjustTree** is invoked which ascends from the leaf L to the root, adjusting covering rectangles and propagating node splits as necessary.

- **Delete:** To delete an object from the R-tree, the deleter invokes the algorithm **FindLeaf** to locate the leaf node L containing the object to be deleted. If the object is not found, the algorithm stops. Otherwise the object is removed from L. If L had just m entries before deletion, the deletion will cause L to become underfull. If that happens, the node is eliminated and each of its entries are reinserted later. Algorithm **CondenseTree** ascends from leaf L to root, adjusting covering rectangles and propagating node elimination upward as necessary.

We refer the interested reader to [9] for a more detailed description of the R-tree.

2.2 The Phantom Problem and Its Solutions

The phantom problem is defined as follows (from the ANSI/ISO SQL-92 specifications [16, 2]): Transaction T1 reads a set of data items satisfying some `<search condition>`. Transaction T2 then creates data items that satisfy T1's `<search condition>` and commits. If T1 then repeats its scan with the same `<search condition>`, it gets a set of data items (known as "phantoms") different from the first read. Phantoms must be prevented to guarantee serializable execution. Object level locking *does not* prevent phantoms since even if all objects currently in the database that satisfy the search predicate are locked, concurrent insertions into the search range cannot be prevented. These insertions may be a result of insertion of new objects, updates to existing objects or rolling-back deletions made by other concurrent transactions.

One solution to the phantom problem is *predicate locking.* In this approach, transactions acquire locks on predicates instead of objects. For example, a range scan that accesses all employees in the database whose salary is between 10K and 20K will acquire a S lock on the predicate: $10K \leq emp.salary \leq 20K$. Locks on two predicates p_1 and p_2 conflict if the predicate (p_1 **and** p_2) is satisfiable, i.e. there may exist an object that satisfies both predicates. Thus, a transaction wishing to insert a new employee record whose salary is 11K will acquire a lock on the predicate $emp.sal = 11K$ which conflicts with the predicate associated with the range scan $10K \leq emp.salary \leq 20K$ and will not be permitted. On the other hand, a transaction wishing to insert a new employee whose salary is 30K will be permitted to execute concurrently with the scan.

While predicate locking provides a complete solution to the phantom problem, testing for predicate satisfiability is usually expensive. Even if predicates are simple and their satisfiability can be checked in constant time, the complexity of acquiring a predicate lock is linear in the number of concurrent transactions which is an order of magnitude costlier compared to acquiring object locks that can be set in constant time [6].

To overcome the high cost of predicate locking, commercial data management systems use a technique called *granular locking.* Granular locking is an engineering approach towards implementing predicate locks. The key idea is to divide the predicate space into a set of resource granules that may include or overlap with other resource granules. Transactions acquire locks on granules instead of on predicates. Transactions either acquire locks in *Shared* mode (S lock) or *eXclusive* mode (X lock). The lock protocol must guarantee that if two transactions request conflicting mode locks on predicates p and p' such that $p \wedge p'$ is satisfiable, then the two transactions will request conflicting locks on at least one granule in common. Granular locks can be set and release as efficiently as object locks. For these reasons, all existing commercial DBMSs use granular locking in preference to predicate locking. We refer interested readers to [6] for a more detailed comparison between the two approaches.

An example of a granular locking approach is *multi-granularity locking* (MGL) [15]. MGL exploits additional lock modes called *intention* mode locks which represent the intention to set locks at finer granularity[15]. An intention mode lock on a node prevents other transactions from setting coarse granularity (i.e S or X) locks on that node. (see the lock compatibility matrix shown

Lock Mode	IS	IX	S	SIX	X
IS	√	√	√	√	
IX	√	√			
S	√		√		
SIX	√				
X					

LOCK MODE	PURPOSE
S	Shared Access
X	Exclusive Access
IX	Intention to set shared or exclusive locks at finer granularity
IS	Intention to set shared locks at finer granularity
SIX	A course granularity shared lock with intention to set finer-granularity exclusive locks (union of S and IX)

Table 2.1: Lock mode compatibility matrix for granular locks.

in Table 2.1). MGL follows the DAG locking protocol as discussed in [6].

Application of MGL to the key space associated with a B-tree is called *key range locking*(KRL). Let $(k_1, k_2, ..., k_n)$ be the ordered list of attribute values present in the B-tree such that $k_i < k_{i+1}$. In KRL, the semi-open ranges $(k_i, k_{i+1}]$ serve as the lockable granules. A scan acquires locks to completely cover its query range. Similarly, a transaction that wishes to insert, delete, or update an object that lies in a given range, acquires an IX lock on the range which denotes its intention to change an object in that range. Dynamic key range schemes are more adaptive to the changes in the key space over time and provides a higher degree of concurrency. However, since the granules may dynamically change, the lock protocols are significantly more complex. Further details about granular locking and KRL can be found in [6].

2.3 Can KRL be Applied to R-trees?

KRL cannot be applied for phantom protection in R-trees since it relies on the total order over the underlying objects based on their key values. While such a total order exists for a single dimensional data, no such natural order exists over multidimensional data [23]. Imposing an artificial total order (say a Z-order [21]) over multidimensional data to adapt the key range idea for phantom protection is unnatural and will result in a scheme with a high lock overhead and a low degree of concurrency. The reason is that the protection of a multidimensional region query from phantom insertions and deletions will require accessing additional disk pages and locking objects which may not be in the region specified by the query (an object will be accessed as long as it is within the upper and the lower bounds in the region according to the superimposed total order) [14]. This severely limits the usefulness of the multidimensional access method, essentially reducing it to a single dimensional access method with the dimension being the total order.

2.4 Desiderata

Since KRL cannot be applied to R-trees, new techniques need to be developed for preventing phantoms in R-trees. This thesis presents a solution based on dynamic granular locking. The main challenges are:

1. Defining a set of lockable granules [1] over the multidimensional key space such that they

    - dynamically adapt to the key distribution
    - fully cover the entire embedded space
    - are as fine as possible so as to maximize concurrency without causing high lock overhead

    In KRL, the key ranges are used as lockable granules. The key ranges satisfy all the three criteria above. On the other hand, in an R-tree, the objects themselves do not fully cover the embedded space. Moreover, object ranges cannot be defined as the embedded space is multidimensional.

2. Easy mapping of a given predicate onto a set of granules that needs to be locked to scan the predicate. Subsequently, the granular locks can be set or cleared as efficiently as object locks using a standard lock manager.

3. Overlap of granules. This problem does not arise in KRL since the key ranges are always mutually disjoint. On the other hand, due to the spatial data overlap, the granules in an R-tree may overlap with each other. The overlapping between the granules complicates the lock protocol since a lock on a granule may not provide exclusive coverage on the space covered by the granule.

[1] In this thesis, we use the term "granules" to mean lock units – resources that are locked to ensure isolation and in not in the sense of granules in "granule graph" of MGL [6].

In our approach, we define a lockable granule associated with each leaf node of the R-tree. They are referred to as *leaf* granules or *tree* granules. The BR of the leaf node defines the lock coverage of the leaf granule. The granules dynamically grow and shrink with insertions into and deletions from the R-tree, thus adapting to the distribution of the objects. Since the leaf level BRs alone may not fully cover the embedded space, we define additional granules called *external* granules, one for each non-leaf node in the tree. The leaf granules together with the external granules fully cover the embedded space. The granules defined above (1) cover the entire space (2) adapt to the key distribution and (3) as argued in Section 3.1, are optimized for high concurrency.

Based on the above granules, lock protocols for various operations have been developed. The developed protocols guarantee phantom protection even when there is mutual overlap among the granules. Finally, the number of locks acquired per operation is low - searchers need to acquire commit duration shared locks on all overlapping granules to protect the search range whereas the inserters and deleters need to acquire just one commit duration lock. The page ids of the leaf nodes are the resource ids used to lock the leaf granules. For external granules, the page ids of the non-leaf nodes are the resource ids. Thus, a logical range can be easily transferred into a sequence of purely physical locks which can be set and checked very efficiently.

Chapter 3

The Granular Locking Approach to Phantom Protection in R-trees

In this chapter, we present the dynamic granular locking approach to phantom protection in R-trees. This is the main contribution of this thesis. In developing the algorithms, we assume that transactions may request the following operations over the R-tree:

- **Insert**: inserts an object into the R-tree,
- **Delete**: deletes an object from the R-tree,
- **ReadSingle**: retrieves a single specific object based on the indexed attributes (also called point search),
- **ReadScan**: retrieve all objects overlapping with a given region (also called region or range search),
- **UpdateSingle**: similar to ReadSingle except that it may modify the qualifying object,
- **UpdateScan**: similar to ReadScan except that it may modify some or all of the qualifying objects.

We assume that an update operation (either UpdateSingle or UpdateScan) can modify only the non-indexing attributes of a qualifying object i.e. attributes not indexed by the R-tree. It can not modify the indexing attributes of the object since that may cause relocation of the object in the R-tree. Such an operation is modeled by the deletion of the original object followed by the insertion of the modified object into the R-tree. In presenting the solution to the phantom problem, we describe the locks that need to be acquired by transactions for each of the above operations. The exact algorithms used to acquire the necessary locks are included in the appendix. The lock protocols assumes the presence of a the standard lock manager (LM). We assume that the LM supports the following lock options [18]:

- *conditional* lock requests which means that the requester is not willing to wait if the lock is not grantable immediately
- *unconditional* lock requests which means that the requester is willing to wait until the lock becomes grantable.

Furthermore, locks can be held for different durations [18]:

- *instant duration* locks which are not actually granted, but the LM has delays (assuming it is an unconditional request) returning the lock call with the success status until the lock becomes grantable.
- *short duration* locks which are released immediately after the operation is over, typically long before the transaction termination.
- *commit duration* locks which are released only when the transaction terminates i.e. after commit or rollback is completed.

Finally, in presenting the lock requirements of various operations for phantom protection, we assume the presence of some protocol for ensuring the physical consistency of the tree structure in presence of concurrent operations. The approach developed in [13] can be used for this purpose. Obviously such a protocol needs to be combined with our approach to phantom protection for the complete solution of concurrency control in R-trees. We do not attempt such an integration in this thesis and restrict ourselves to addressing only the problem of phantoms in R-trees.

3.1 Partitioning the Space into Granules

One of the major challenges in developing a granular locking solution to phantom problem in R-trees is to partition the embedded space into lockable granules that dynamically adapts to the

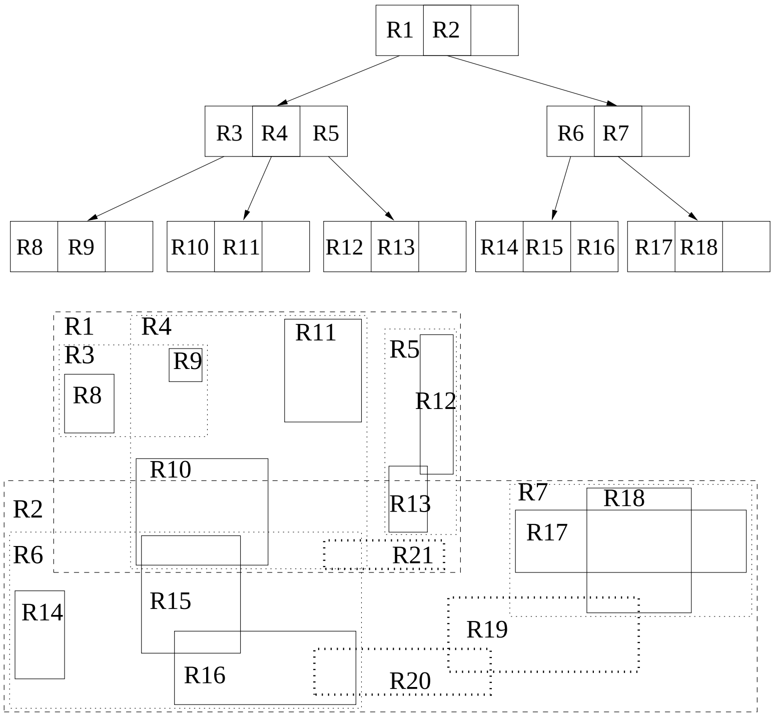

Figure 3.1: An R-tree with search predicate ($R19$)and insertion predicates ($R20$ and $R21$).

distribution of objects. To maximize concurrency, the granules should be as fine as possible. In KRL, the semi-open key ranges are treated as lockable granules. Key ranges cannot be defined in R-trees since the key space is multidimensional and there exists no linear ordering in multidimensional space. We define a lockable granule associated with each leaf node of the R-tree. We refer to these granules as **leaf granules** or **tree granules**. We denote the leaf granule associated with leaf node L as $TG(L)$. The lock coverage of $TG(L)$ is defined by $BR(L)$ i.e. the smallest rectangle that covers all the objects stored in L. Formally, lock coverage of $TG(L)$ is defined as

$$\text{Lock Coverage}(TG(L)) = BR(L) = \begin{cases} S \text{ if } L \text{ is the root, where } S \text{ is the entire embedded space} \\ E_L.I \text{ otherwise, where } E_L \text{ is } L\text{'s entry in the parent of } L \end{cases} \tag{3.1}$$

The page id of the leaf node is used as the resource id to lock the corresponding leaf granule. For example, in the R-tree shown in Figure 3.1 there are 5 leaf granules: $TG(R3)$, $TG(R4)$, $TG(R5)$, $TG(R6)$ and $TG(R7)$ corresponding to the leaf nodes $R3$ to $R7$. For simplicity, we sometimes denote the leaf granule $TG(R)$ as simply R, the meaning should be clear from the context.

If use only the leaf granules as the only lockable granules, the set of lockable granules may not cover the embedded space. The reason is that the R-tree is not a space-covering index structure i.e. the R-tree partitions may not cover the entire embedded space. This presents a problem as the set of granules may not be able to properly protect search predicates resulting in phantoms. To see this, consider again the tree shown in Figure 3.1. Let transaction $t1$ request a scan on the rectangle $R19$ which only overlaps with granule $R7$. An S lock on $R7$ will not prevent the insertion of an object $R20$ by another transaction $t2$ since $R20$ does not overlap with $R7$. Insertion of $R20$, however, should not be permitted since it overlaps with the region being scanned by $t1$ (that is, $R19$). This problem would not arise if the set of granules fully covered the embedded space and $t1$ had acquired locks on enough granules to cover its search predicate.

A simple solution to cover the entire embedded space is to maintain a single extra lockable granule which covers the space that is not covered by the R-tree leaf granules. This solution is impractical since the extra granule will become a hot spot leading to very low concurrency. Non conflicting predicates with conflicting lock modes that overlap with the space not covered by the leaf granules would not be able to execute concurrently as they would require conflicting locks on the extra granule. For example, in Figure 3.1, a read scan over rectangle $R19$ and an insertion of rectangle $R21$ would not be granted locks concurrently although the rectangles do not intersect.

We present a scheme that partitions the non-covered space into a set of granules. We refer to them as **external** granules. An external granule is associated with each non-leaf node of the R-tree whose shape and size is determined by the R-tree partitionings. Let T be a non-leaf node of the R-tree and $BR(T)$ be the BR of T. Let $ext(T)$ denote the external granule associated with T. The lock coverage of $ext(T)$ is defined as follows. Let $E_1, E_2, ...E_n$ be the entries in T. Each E_i has two elements: *(I, child-pointer)* where *child-pointer* is the address of a lower node in the R-tree and I covers all rectangles in the lower node's entries as discussed in Section 2.1. The lock coverage of $ext(T)$ is:

$$\text{Lock Coverage}(ext(T)) = BR(T) - \bigcup_{i=1}^{n} E_i.I \qquad (3.2)$$

In other words, the external granule of a non-leaf node T covers the space covered by the BR of

T but not by the BRs of any of its children. The page id of the non-leaf node T is used as the resource id to lock $ext(T)$. The mapping of granules to the resource ids is therefore simple for both types of granules. This makes it easy to transfer a logical range to a sequence of purely physical locks which can be set and checked efficiently by a standard lock manager.

In Figure 3.1, there are 3 external granules, one associated with each non-leaf node. The external granule associated with the root (i.e. $ext(root)$) node covers the space $S - (R1.I \cup R2.I)$ where S is the total embedded space. The external granules associated with the nodes bounded by $R1$ and $R2$ (i.e. $ext(R1)$ and $ext(R2)$) cover spaces $R1.I - (R3.I \cup R4.I \cup R5.I)$ and $R2.I - (R6.I \cup R7.I)$ respectively. Note that the external granules along with the leaf granules completely cover the embedded space. For example, for the R-tree in Figure 3.1, the union of $ext(root)$, $ext(R1)$, $ext(R2)$, $R3$, $R4$, $R5$, $R6$ and $R7$ is the entire embedded space S.

Note that the partitioning strategy described above could alternatively be applied to the object level instead of the leaf node level. For example, for the R-tree in Figure 3.1, the extents of the objects $R8$ to $R18$ could be considered as the lockable granules. Unlike the previous partitioning strategy where external granules are only associated with non-leaf nodes, in this strategy, there are external granules associated with all index nodes i.e. both non-leaf and leaf nodes. In this partitioning too, the set of granules cover the entire embedded space. In this case, the granules are finer than the ones in which the leaf level BRs are used as tree granules. Although the granules are finer, this scheme does not necessarily result in an improvement in concurrency. The reason is that if a scan predicate overlaps with a leaf node L, it typically spans more than one object extent in that node and hence would overlap with $ext(L)$. As a result, the scan operation would almost always need to lock $ext(L)$. The same is true for other operations. This means that conflicting operations within the same leaf node would almost always conflict on the lock on the external granule associated with that node. Having finer granules would increase the lock overhead without providing much additional concurrency. Hence, we support the leaf level BRs as the leaf granules since it provides as high concurrency as supporting objects extents as granules but with considerably lower lock overhead.

3.2 Problems Due to Growing Granules

Once the embedded space has been partitioned into a set of lockable granules, we now discuss how the granular locking protocol can be applied to the set of granules. Following the principles of granular locking, each operation requests locks on enough granules to guarantee that any two conflicting operations request conflicting locks on at least one granule in common. We present two possible strategies that satisfy the above criteria:

- **Overlap-for-Search and Cover-for-Insert (OSCI) Policy**: This policy requires the search operation to acquire S locks on all granules that overlap with the predicate being scanned and the insert,delete and update operations to acquire IX locks on a minimal set of granules sufficient to fully cover the object (in addition to the X lock on the object itself). In this strategy, the insertion of an object that overlaps with the search region of a query is not permitted to execute concurrently, thereby preventing phantoms from arising. For example in Figure 3.1, a transaction wishing to insert rectangle $R21$ acquires IX locks on granules $ext(R1)$ and $R4$ [1]. A searcher wishing to scan predicate $R19$ acquires S locks on $ext(R2)$ and $R7$.

- **Cover-for-Search and Overlap-for-Insert (CSOI) Policy** This policy requires the insert, delete and update operations to acquire IX locks on all overlapping granules and the search operation to acquire S locks on the minimal set of granules that cover the scan predicate.

The above policies guarantee phantom protection when the operations do not dynamically change the lock coverage of the granules. The formal proofs are included in the Appendix.

Since the lock coverage of the granules depend on the 'structure' of the R-tree, operations that change the structure (e.g., insertions and deletions) *can* dynamically modify the granules. Under these situations, neither of them are sufficient to prevent phantoms. We illustrate it by a couple of examples. First let us consider the OSCI policy. In Figure 3.2(a), $R1$ and $R2$ are two leaf granules within the same BR R. Assume a transaction $t1$ arrives to scan the rectangle $R3$. It acquires an S lock on $R1$ and proceeds with the scan. Then another transaction $t2$ arrives to insert rectangle

[1] Since both $R1$ and $R2$ fully covers $R21$, either path can be selected. If $R2$ was selected, then the transaction would acquire IX locks on $ext(R2)$ and $R6$

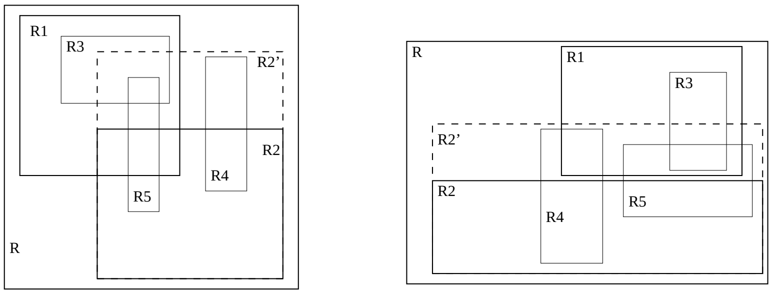

Figure 3.2: Insertions cause leaf granules to grow into other tree granules. (a) shows the problem with the cover-for-insert and overlap-for-search policy (b) shows the problem with overlap-for-insert and cover-for-search policy.

$R4$, acquires IX locks on $R2$ and $ext(R)$, an X lock on $R4$ and inserts $R4$. As a result, $R2$ grows to $R2'$. Then $t2$ commits and releases all its locks. Now a third transaction $t3$ arrives to insert rectangle $R5$. $R5$ is completely covered by $R2'$ and hence if the above protocol is followed, $t3$ just needs to acquire an IX lock on $R2'$. $t3$ gets the lock and proceeds with insertion which should have been prevented because if $t1$ now repeats its scan, it would find $R5$ has appeared from nowhere.

The problem in the above example arises since growth of the BR $R2$ to $R2'$ causes $t1$ to lose its S lock on part of its scan predicate $R3$ (specifically, on the region $R3 \cap R2'$). Since the lock protocol developed in the previous section follows the cover-for-insert and overlap-for-search policy, for $t1$ to have an S lock on $R3$ *after* the rectangle $R2$ had grown to $R2'$, it requires S locks on both granules $R1$ and $R2'$. An S lock on $R1$ alone does not provide the necessary protection against phantoms. Since a search operation does not cause granules to either grow or shrink, at first glance it may appear that the CSOI policy may resolve the problem. Indeed, if the CSOI policy is used, the above discussed situation will not arise. But even the CSOI policy does not prevent phantoms from arising as shown in the next example.

In Figure 3.2(b), $R1$ and $R2$ are two leaf granules within the same BR R. Assume a transaction $t1$ arrives to insert rectangle $R3$. It acquires an IX lock on $R1$ as $R3$ overlaps only with $R1$, an X lock on $R3$ and proceeds with insertion. Then another transaction $t2$ arrives to insert rectangle $R4$. $R4$ overlaps with $R1$, $R2$ and $ext(R)$. So $t2$ acquires IX locks on $R1$, $R2$ and $ext(R)$, an X lock on $R4$ and inserts $R4$. As a result, $R2$ grows to $R2'$. Then $t2$ commits and releases all its locks. Now a third transaction $t3$ arrives to scan rectangle $R5$. $R5$ is completely covered by $R2'$ and hence $t3$

just needs to acquire an S lock on $R2'$. $t3$ gets the lock and proceeds with the scan which again should have been prevented because if $t1$ aborts and $t3$ repeats its scan it will find the object $R3$ has disappeared.

To identify the lock requirements of the operations when the granules can dynamically change, we need to carefully analyze how operations modify granules and how the modifications affect the lock coverage of transactions that had previously acquired locks. We conduct such an analysis for the OSCI policy. This policy is preferred over the CSOI policy for R-trees since searchers in R-trees anyway follow all overlapping paths while inserters follow single paths. The CSOI policy may be better for index structures where inserters follow all overlapping paths and searchers follow only enough paths to cover its predicate. The R+-tree is an example of such an index structure [24]. We assume that the OSCI strategy is followed for all discussions in the rest of the paper. In conducting the analysis on how granule changes affect lock coverage, we differentiate between the leaf granules and the external granules.

The operations that may modify the current set of granules are the insert and the delete operations. In the simple case, an insertion may cause the BR of the leaf node into which the object is inserted, referred to as the *target node*, to grow. This causes the corresponding leaf granule, referred to as the *target granule*, to grow. When the BRs are adjusted bottom-up, the BRs of ancestors of the target node may also grow affecting the lock coverages of the corresponding external granules. An insertion may also cause node splits which may propagate upwards causing the higher level nodes to split. Node splits also affect the lock coverage of external granules.

Similar to an insertion, a deletion may also cause the granule associated with the leaf node from which the object is deleted as well as the external granules associated with the ancestors of the node to change. Occasionally, a deletion may cause node elimination and thus causing the granule associated with the node to disappear. As a result, the external granules associated with the ancestors of the eliminated node may also change. Also, the entries of the eliminated node will get reinserted into some other part of the tree. This may cause the granules associated with the nodes into which reinsertion takes place and the external granules associated with their ancestors to change.

In the next 5 subsections, we describe how each of these operations affects the lock coverage of other transactions and develop solutions to prevent phantoms that may arise due to the changes in the lock coverage.

3.3 Search

The search operation, also referred to as range search or readscan operation, acquires S lock on all granules, both leaf and external, overlapping with the search region. The search operation does not modify the lock coverage of any granule. The concurrent search algorithm is shown in the appendix (Table B.1).

3.4 Insertion

We first consider the case when the insertion does not cause any node split. Let g be the target granule i.e. the granule in which the insertion takes place. Insertion of an object in g may cause the following two things to happen:

- **Growth into other Leaf Granules:** The leaf granule g grows into some other leaf granule(s) causing the overlap between the two granules to increase. For example, in Figure 3.2(a), the growth of $R2$ to $R2'$ increases the overlap between $R1$ and $R2$.

- **Growth into External Granules:** The granule g grows into the external granule associated with the parent of g causing the external granule to shrink in size. As the BRs are adjusted from the leaf to the root, the external granules associated with the non-immediate ancestors of g may change as well. For example, in Figure 3.2(a), the growth of $R2$ to $R2'$ shrinks the external granule $ext(R)$ from $R - (R1 \cup R2)$ to $R - (R1 \cup R2')$. Notice that only the external granules associated with the ancestors of the granule into which the insertion took place can change.

Both the above situations could result in phantoms. In this section, we consider the two situations one by one, show how they can give rise to phantoms and present lock protocols to prevent them from arising.

3.4.1 Growth into Other Leaf Granules

We start with the first situation. The two examples discussed earlier illustrate this situation. When the target granule g grows into a leaf granule h, a transaction t holding an S lock on h prior to the growth may lose its lock coverage on the portion of h into which g grew. The reason is that to acquire an S lock on a region, a transaction needs to S lock all granules that overlap with the region. Since after the growth, g may also overlap with the search region of t, a lock on h alone may not provide the required protection from future insert or delete operations and may result in phantoms.

One approach to preventing phantoms is for all the locks held by transactions on granule h to be inherited by the granule g that is growing, where h is a leaf granule into which g grows. This approach has several problems with regard to its implementation. We need to access to the transaction table and transaction lock lists to find out the transactions that hold locks on h. This is also expensive since the search has to be repeated for each leaf granule h into which g grew. Lock inheritance is also difficult to implement using the standard API of a lock manager. We do not consider this approach any further since we desire the lock protocols to be able to use the LM with just the standard API and without requiring access to its private functions and/or internal data structures.

We solve the problem by slightly modifying the OSCI policy. In the modified policy, we distinguish between the two types of insertions and handle them differently. Let g be the target granule and O be the object being inserted.

- *Insertions that do not cause the target granule g to grow*: These insertions follow the normal OSCI policy i.e. they acquire IX locks on the minimal set of granules covering O. Since O does not cause g to grow, g fully covers O. So the inserter just needs two locks: (1) a commit duration IX lock on g and (2) a commit duration X lock on O itself.

- *Insertions that cause g to grow (to say g')*: These insertions require more locks than the normal OSCI policy. The inserted acquires (1) a commit duration IX lock on g (2) a commit duration X lock on O and (3) short duration IX locks on *all* granules into which g grew i.e. all granules overlapping with $(g' - g)$. The short duration locks in (3) ensures that there no searcher can lose their lock coverage due to the growth of g. Let us revisit the situation in Figure 3.2(a). Since $t2$ causes $R2$ to grow, $t2$ needs to acquire the extra short duration locks on all leaf granules overlapping with $(R2' - R2)$. Since $R1$ overlap with $(R2' - R2)$, $t2$ requests a short duration lock on $R1$ which will block $t2$ till $t1$ terminates (since $t1$ has an S lock on $R1$). This implies that in the new protocol, no searcher can lose its lock and hence future inserters (like $t3$ in Figure 3.2(a)) cannot give rise to phantoms. Note that the locks in (3) along with lock in (1) satisfy the full cover criteria of the OSCI policy. The locks in (3) can be short duration since: (i) Since only old searchers can lose their locks, we do not need more than short duration locks to prevent the loss of locks. (ii) They are not for full cover either after the operation is over since g has now grown and the lock in (1) alone suffices for full cover.

3.4.2 Overhead Estimation

In an R-tree, inserters always follow a single path from the root to the target leaf i.e. it accesses just one node per level. In the above protocol, the inserters that cause granule growth need to access some extra nodes to acquire those short duration locks. The above protocol imposes an overhead on these insertions in terms of disk accesses. The average I/O overhead imposed by the protocol depends on two factors:

- The fraction of inserters that cause granule change. Only these inserters pay the overhead; the others do not pay any overhead.
- The average I/O overhead for those inserters that cause granule change.

We performed some simple experiments to estimate the above factors. We performed the experiments on 2 synthetic datasets:

- *Point dataset*: This dataset consists of 32,000 points uniformly distributed in 2-d space.
- *Rectangle dataset*: This dataset consists of 32,000 uniformly distributed 2-d rectangles. The average extent of the rectangles along each dimension is about 5% of the extent of the total data space along that dimension.

Nature of data	Fanout	Height	Insertions Causing Granule Change	Avg. Disk Accesses Per Insertion Causing Granule Change			Extra I/O Cost Per Insertion Causing Granule Change	Extra I/O Cost Amortized over all insertions
				Level 2	Level 3	Level 4		
Point	50	3	6%	1.567	-	-	0.567	0.034
Rectangle	50	3	8%	1.871	-	-	0.871	0.070
Point	24	4	15%	1.154	1.938	-	1.092	0.164
Rectangle	24	4	19%	1.197	2.702	-	1.899	0.361
Point	12	5	35%	1.012	2.144	2.640	2.796	0.979
Rectangle	12	5	38%	1.033	2.043	3.137	3.213	1.221

Table 3.1: I/O overhead of insertion in the granular locking approach.

For each dataset, we constructed 3 R-trees of different heights by specifying different fanouts. The R-trees have heights 3, 4 and 5 with fanouts 50, 24 and 12 respectively. For each tree, we executed several insertions and measured the %age of insertions that caused granule change. Among the insertions that caused granule change, we measured average number of disk pages accessed at each level in order to identify all leaf nodes in which g grew. The results are summarized in Table 3.1. The levels of the tree are numbered from top to bottom i.e. the root is level 1 and the leaf level is level h (h is the height of the tree). Note that an inserter never needs to access the leaf nodes for acquiring the short duration locks. The number of accesses for the root level is always 1 since there is only 1 root node. Hence the table lists the average number of node accesses for only levels 2 to $(h-1)$. The average extra I/O cost per inserter for any level is the average number of node accesses for that level minus one since the inserter needs to access one node per level anyway. We calculate the average I/O cost per inserter causing granule change by summing the I/O costs per level. We calculate the average I/O cost per inserter (amortized over all inserters) by multiplying the fraction of inserters causing granule change with the average I/O cost for such inserters.

The experiments show that the %age of insertions causing granule change depends on the fanout of the tree. The larger the fanout, the lower the %age of insertions causing granule change. This is expected since the larger fanout implies larger sized granules which in turn means lower probability of insertions changing the target granules and hence lower fraction of inserters causing granule change. The average I/O overhead per inserter causing granule change depends on the height of the tree. It varies between approximately 0.5 to 3 disk accesses per insertion. The overhead will be

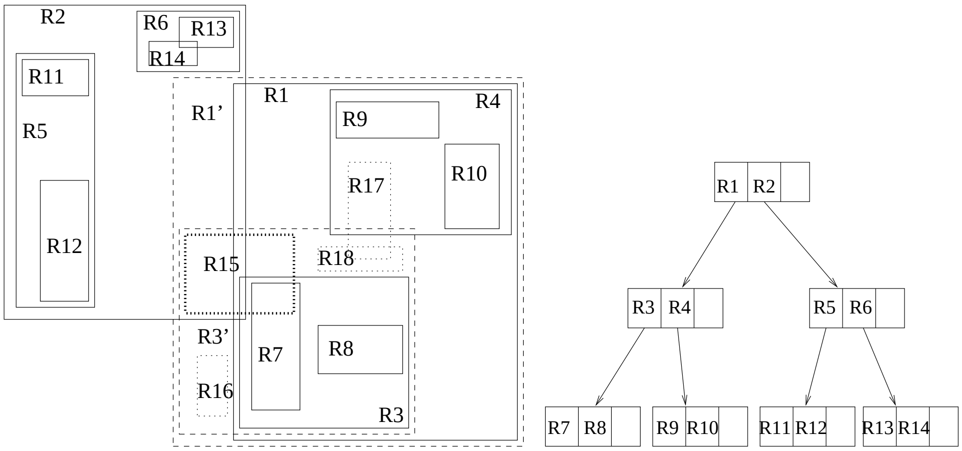

Figure 3.3: Insertions cause external granules to shrink. $R15$ is the object being inserted. R16, R17 and R18 are other predicate rectangles. The R-tree corresponding to the initial spatial distribution of the objects is shown alongside.

lower if we take buffer management into account. According to the 5-minute rule, with a reasonable size buffer, the top 2-3 levels of the R-tree will always reside in memory. So all extra node accesses do not cause extra disk accesses. With a reasonably large fanout (50-100) and decent size buffer, we expect the average I/O overhead per inserter to be less than 0.1 i.e. less than 1 extra disk I/O per 10 insertions.

3.4.3 Growth into External Granules

While the above discussion describes techniques to overcome problems that arise due to the growth of a granule into other leaf granules, phantoms may also arise due to growth of a granule into an external granule causing the latter to shrink. Such a situation is illustrated next.

In Figure 3.3, let us consider a transaction $t1$ wishing to insert rectangle $R15$ into the granule $R3$. $t1$ acquires an IX lock on $R3$ and short duration IX locks on $ext(R1)$ and $ext(R2)$ and proceeds with the insertion. The insertion causes $R3$ to grow to $R3'$ which in turn causes $R1$ to grow to $R1'$. Let us consider an old active transaction $t2$ wishing to insert rectangle $R17$ into granule $R4$. It had acquired an IX lock on $R4$ and a short duration IX lock on $ext(R1)$. Due to growth of $R3$ to $R3'$, $ext(R1)$ shrinks due to which $t2$ loses its short duration lock on $ext(R1)$. After $t1$ commits, if a searcher arrives to scan region $R18$, it will be incorrectly permitted to do so because it only needs an S lock on $R3'$. Similar to inserters, searchers also may lose their lock due to shrinkage of external granules. For example, in Figure 3.3, a searcher scanning region $R16$ loses its lock on $ext(root)$ when $R1$ grows to $R1'$.

To prevent loss of locks held by other transactions, an inserter must ensure that no other transaction is holding any lock on the external granule before growing into it. This can be achieved by acquiring short duration SIX locks (since they conflict with all lock modes except the IS mode which is never used by the protocol) on the external granules that change due to the insertion of the object. [2] In the above example, $t1$ should acquire a short duration SIX lock on $ext(R1)$ before adjusting $R3$ to $R3'$ and a short duration SIX lock on $ext(root)$ before adjusting $R1$ to $R1'$.

While acquiring SIX locks on the external granules prevents other transactions from losing their locks on the external granules, if the inserting transaction t itself was holding an S lock on the external granule that shrinks due to the insertion, it may lose its S lock coverage after the growth. To prevent this, t needs to acquire an S lock on the growing granule g. Acquiring an S lock on g provides t with the same lock coverage it had prior to the growth of g since the amount of coverage lost due to shrinkage of the external granule is fully covered by the growth of g.

Above, we have described how phantoms can arise due to dynamic changes to the granules resulting from insertions and presented strategies to prevent them. To insert an object, since OSCI policy is used, a single commit duration lock on the granule into which the object is inserted is sufficient. After the insertion, the granule grows to cover the inserted object. All the other locks are acquired for a short duration to prevent the potential loss of locks by other operations due to the dynamic changes to the granules resulting from the insertion. The concurrent insertion algorithm is shown in Table B.2.

3.5 Node Split

We now consider the case in which the insertion of an object O by a transaction t into an already full node causes the corresponding granule g to split into granules g_1 and g_2. Since after the split, the IX lock held by t on g is lost, t needs to acquire an IX lock on either g_1 or g_2, whichever contains O. Since t acquires an IX lock on g before the insertion, no other transaction can be holding an S

[2] The maximum number of external granules that can change is equal to the height of the R-tree minus one. The number of external granules that actually change may be less because the change may not propagate all the way up to the root.

lock on g. If t itself was holding an S lock on g, just inheriting the S lock of t on g to g_1 and g_2 will not prevent phantoms since after the split $g_1 \cup g_2$ may not fully cover g. To cover the region originally covered by g, t will need to acquire S locks on g_1 and g_2 after the split as well as S locks on all additional granules that overlap with both with the predicate $g' = g - (g_1 \cup g_2)$ and the search region.The only additional lock it needs to satisfy the above requirement in an S lock on $ext(P)$ where P is the parent of g.

Since before the split the inserter acquires an IX lock on g, other inserters and deleters may be holding IX locks on g which will be lost when g disappears. To prevent this, all transactions holding IX locks on g must acquire IX locks on either g_1 or g_2, whichever is appropriate, after the split. This is sufficient since every insert and delete range protected by the IX lock on g before split will be protected one of the two granules, either g_1 or g_2, after the split. This involves lock inheritance and as mentioned before, is difficult to implement using the standard API of a lock manager. We avoid the situation by requiring the inserter to acquire a short duration SIX lock instead of an IX lock on g if it causes g to split. After the split, the inserter just needs to IX lock on the granule containing O.

The splitting of the granule may propagate upwards causing the higher level nodes to split. When a non-leaf node N splits, the external granule associated with N shrinks in size due to which a transaction holding a lock on the external granule may lose its lock. To prevent this, transactions need to acquire short duration SIX locks on the external granules associated with the parent nodes during split propagation. Since inserters anyway acquire short duration SIX locks on the external granules associated with the ancestors of the granule in which the insertion took place while propagating the changes in BRs (even if there is no node split), no additional locks are required on the external granules when the insertion causes the node to split. However, if the transaction itself was holding an S lock on the external granule $ext(N)$ that shrinks due to the split of N to N_1 and N_2, it may lose its S lock coverage after the split. So it needs to acquire S locks on $ext(N_1)$, $ext(N_2)$ and $ext(P)$ where P is the parent of N to preserve the lock coverage.

The concurrent insert algorithm is shown in the appendix (Table B.2).

3.6 Logical Deletion

Similar to insertion, to delete an object O an IX lock on the region that covers the deleted object is required. However, unlike insertion, (in which the granule where the object is inserted grows and covers the inserted object), the granule g from which the object is deleted may shrink (to g') and hence not cover O after adjustment.

To protect the delete region, the deleter needs to acquire commit duration IX locks on the minimal set of granules $\mathcal{C}$ sufficient to fully cover the deleted object. The set $\mathcal{C}$ includes g and the minimal set of additional granules whose union fully covers the predicate $O \cap (g - g')$ (as g only covers $O \cap g'$ after the operation). Acquiring locks on each granule in $\mathcal{C}$ requires a top-down tree-traversal. Further, multiple commit duration locks need to be acquired. For this reason, we do not consider this approach any further.

Instead, deletes are performed logically. The deleter needs to acquire only a commit duration IX lock on the granule that contains the object and an X lock on the object itself. If the transaction aborts, the R-tree remains unchanged. On the other hand, if the deleter commits, the physical deletion of the object from the R-tree is executed as a separate operation.

If the transaction requests deletion of an object that does not exist, other transaction wishing to insert the same object should be prevented as long as the deleter is active. For this purpose, the deleter acquires S locks on all overlapping granules just like a ReadScan operation with the object as the scan predicate.

3.7 Deferred (Physical) Deletion

The deferred delete operation physically removes the logically deleted object from the R-tree and adjusts the BRs of the ancestors. To physically delete an object from a granule g, the transaction requires an IX lock on g to prevent searchers having S locks on g from losing their lock coverage. If the deferred delete operation is executed as part of the same transaction as the corresponding logical delete, it already has a commit duration IX lock on g. Otherwise, the lock needs to be reacquired. Deletion of an entry from the node may cause the node becoming underfull in which case the node is eliminated and its entries are later reinserted. In this case, not only searchers but also transactions holding IX locks on g may lose their lock coverage due to elimination of g. To prevent this, if the deletion causes the leaf node to become underfull, the deleter acquires an instant duration SIX lock on g before eliminating g. In either case, since BR of the ancestor nodes of g (granule from which the object is physically removed) needs to be adjusted, the external granules associated with the ancestors may shrink. To prevent loss of lock coverage by other transactions due to this shrinkage, similar to insert, instant duration SIX locks are acquired on external granules that shrink when the BRs are adjusted.

If deletion of the object had caused the node to become underfull and the underfull node eliminated, the remaining entries of the eliminated node are reinserted. The node elimination may propagate up causing elimination of higher level nodes. In such a case, the entries of the eliminated node must be placed higher in the tree, so that leaves of their dependent subtrees will be on the same level as leaves of the main tree [9, 3]. Instant duration SIX lock is required on the external granule if either the node is eliminated (external granule disappears) or the BR of the node needs to be adjusted (external granule shrinks).

When the entries are reinserted, the lock requirements depend on whether the entry E being reinserted is a data object (if the eliminated node was a leaf node) or an index node (if the eliminated node was a non-leaf node). In both cases, similar to an ordinary insert operation, an IX lock needs to be acquired on the granule in which the reinsertion takes place (along with the short duration IX locks on all overlapping granules and short duration SIX locks on the external granules associated with the parent nodes if the reinsertion causes the granule boundary to change). If the entry E is an index node, the external granule g associated with the node into which E is inserted may shrink causing a transaction holding a lock on g to lose its lock. To ensure that no transaction loses its lock coverage, a short duration SIX lock is required on g. On the other hand, if E is an object, it will be inserted into a leaf granule which can only grow due to the insertion. Since no transaction loses its lock in this case, the SIX lock is not required for object reinsertion.

The concurrent logical and deferred delete algorithms are shown in the appendix (Tables B.5 and B.7).

3.8 Other Operations

The ReadSingle operation acquires an S lock on the object. The UpdateSingle operation acquires an IX lock on the granule containing the object and an X lock on the object. The UpdateScan operation acquires S lock on all overlapping granules and the same locks as an UpdateSingle operation for every individual object updated (see appendix (Table B.9 for the concurrent UpdateScan algorithm). The lock requirements for the various operations is summarized in the Table 3.2.

Operation	Overlapping granules ($\mathcal{E}$)	Minimal covering granules ($\mathcal{C}$)	Granule g containing object	Object (O)	Others ($\mathcal{X}$)
Insert (No split or granule change)	None	None	IX	X	None
Insert (Granule change)	Short duration IX	None	IX	X	Short duration SIX on $ext(P^*)$ †
Insert (Node split)	None	None	*Before split*: Short duration SIX on g *After split*: If splitting node N is leaf, IX on either $TG(N1)$ or $TG(N2)$, whichever contains O. If trans. itself holding S lock on N, acquire S locks to $TG(N1)$, $TG(N2)$ and $ext(P)$ if N is leaf; to $ext(N1)$, $ext(N2)$ and $ext(N)$ if N is non-leaf .	X	Short duration SIX on $ext(P^*)$ †
Delete (Logical)	None	None	IX	X	None
Delete (Deferred)	None	None	Short duration IX if g does not become underfull Short duration SIX if g becomes underfull	X	Short duration SIX on $ext(P^*)$ and locks for reinsertion of orphan entries
ReadSingle	None	None	None	S	None
ReadScan	S	None	None	None	None
UpdateSingle	None	None	IX	X	None
UpdateScan	S	None	IX for each object updated	None for read X for update	None

Table 3.2: Lock protocols and lock modes required for various operations in the dynamic granular locking approach. P^* denotes all the ancestors of g whose external granule changes due to the operation. Since $g \in \mathcal{C} \subseteq \mathcal{E}$, the locking of $\mathcal{E}$ implies locking of $\mathcal{C}$ and locking of $\mathcal{C}$ implies locking of g. † If the inserter itself was holding an S lock on $ext(P)$, the growing/splitting granule(s) must inherit the lock.

Chapter 4

Experimental Evaluation

We performed several experiments to evaluate the performance of the granular locking protocol proposed in this thesis under various degrees of system loads. In this chapter, we discuss our implementation of the protocols followed by the performance results. We concluded the chapter with a comparison with the predicate locking approach proposed in [13].

4.1 Implementation

4.1.1 Implementation of Protocols

We implemented the complete granular locking protocol (referred to as the *GL* protocol) as described in this paper. To evaluate the performance of the protocol, we also implemented the pure predicate locking protocol (referred to as the *PL* protocol) to serve as the baseline case. In PL, each search operation checks its predicate against the objects of the insert/delete/update operations of all currently executing transactions. If there is any conflict, it blocks on that transaction by requesting an S lock on that transaction ID, assuming that every transaction acquires an X lock on its own ID when it starts up. Otherwise it proceeds with the search. Similarly, each insert/delete/update operation checks its object against the predicates of the search operations of all currently executing transactions and in case of a conflict, blocks on the conflicting transaction.

4.1.2 Construction of R-trees

We conducted our experiments on two different R-trees constructed over the following two datasets:

- The **SEQUOIA Point dataset:** is the 2-d point data set of the Sequoia 2000 benchmark

Parameters	Meaning
MPL	multiprogramming level
Transaction Size	the number of operations per transaction
Write Probability	the fraction of operations in a transaction that are writes (i.e. inserts)
Query Size	the average selectivity of a search operation
External Think Time	mean time between transactions
Restart Delay	mean time after which an aborted transaction is restarted

Table 4.1: Workload parameters.

[25]. It contains locations(easting and northing values) of 62,556 California places extracted from the US Geological Survey's Geographic Names Information System (GNIS)). The points are geographically distributed over a 1046km by 1317km area.

- The **SEQUOIA Polygon dataset:** is the 2-d polygon data set of the Sequoia 2000 benchmark. It contains 79,607 polygons (delimited by 3,997,756 points) of homogeneous land use, extracted from the US Geological Survey's Geographical Information Retrieval and Analysis System (GIRAS). We approximated the polygons by their their least bounding rectangles. The rectangles are geographically distributed over a 880 km by 1370 km area.

We downloaded the above datasets from the Sequoia 2000 Project's FTP site (http://s2k-ftp.cs.berkeley.edu:8000/sequioa/benchmark/).

We set aside some points/rectangles (by random choice) from the above data files for insertion into the R-tree during the run of transactions. The searches to be executed during the run are generated by first randomly choosing the query anchor from the data file and generating a bounding box by choosing a proper side length needed to obtain desired search selectivity. For point data, the randomly chosen point itself is the query anchor while for polygon data, we use the center of the bounding rectangle as the query anchor. The set-aside points/rectangles and the queries are stored in two separate files which are used by the workload generator.

We created the R-trees by bulkloading the remaining points. The two R-trees are described below:

- *Point R-tree:* constructed on 56,655 2-d points with 2K page size (fanout 102, 821 nodes).
- *Polygon R-tree:* constructed on 71,229 2-d rectangles with 2K page size (fanout 102, 1061 nodes)

Since the the data sets are small, we choose comparatively small page sizes to make the R-trees of significant size (at least 3 levels).

4.1.3 Workload Generator and the Lock Manager

The workload generator (WG) generates a workload based on the input parameters shown in Table 4.1. The WG assigns some search operations (from the bounding box query file) and some insertion operations (from the set-aside point/rectangle file) to each transaction. Each transaction executes as a separate thread. We use the Pthread library (Solaris 2.6 implementation) for creating and managing the threads [19]. One thread only executes one transaction: it is created at the beginning of the transaction and is terminated when the latter commits. The WG maintains the MPL at the specified value by using an array of flags (MPL number of them): when a thread finishes, it sets a flag. The main WG thread constantly polls on this array and when it detects the setting of a flag, it starts a new thread and assigns the next transaction to it. The thread waits for some time (external think time) and starts executing the transaction: it executes one operation after another on the R-tree following the lock protocols. If any lock request returns an error (due to a deadlock or a timeout), the transaction aborts. If it aborts, it is re-executed within the same thread after a certain restart delay (each transaction remembers its constituent operations till it commits for possible re-execution). Our implementation of the WG consists of 3 main C++ classes (TransactionManager, Transaction and Operation). The TransactionManager class also maintains the global statistics of the run (e.g., throughput, number of locks acquired, number of aborts etc.) which are used to measure the performance of the various protocols. Although the other 4 simulation parameters are varied, we fix the external think time to 3 seconds and the restart delay to 3 seconds for all our experiments. Also, for the two R-trees, the buffer sizes are set such that about 75% of the pages fit in memory.

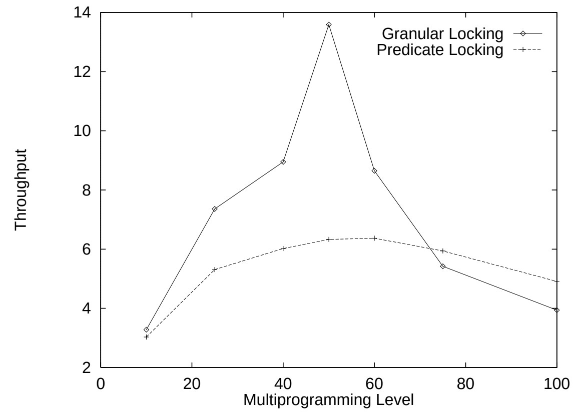

Figure 4.1: Throughput at various MPLs for 2-d point data (write probability=0.2, transaction size=10, query selectivity=0.1%).

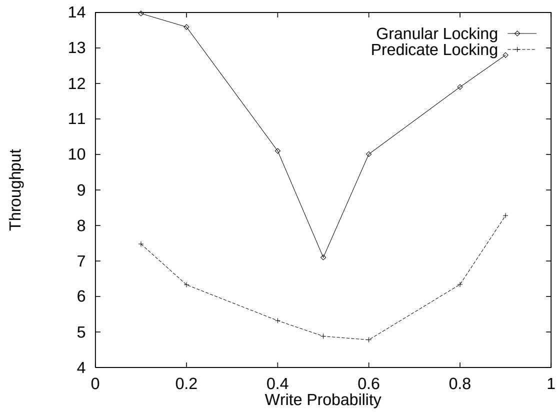

Figure 4.2: Throughput at various mixes of reads and writes for 2-d point data (MPL=50, transaction size=10, query selectivity=0.1%).

For the lock manager (LM) implementation, we reused most of the LM code of MiniRel system obtained from the University of Maryland. The LM code closely follows the description in [6].

All experiments were performed on a Sun Ultra Enterprise 3000 Server running Solaris 2.6 with two 167MHz CPU, 512MB of physical memory and several GB of secondary storage.

4.2 Experimental Results

We conducted experiments to evaluate the performance of the granular locking protocol for R-trees under various system loads. Performance is measured using throughput i.e. the ratio of the total number of transactions that completed during the period when the transactions ran at full MPL (ignoring the starting phase and the dying phase when the MPLs are lower) to the total duration of the full-MPL phase [1]. Figures 4.1 shows the throughput of GL and PL protocols at various MPLs for the 2d point dataset. Initially, the throughput increases with the MPL as the system resources were underutilized at low MPLs. For GL, the throughput reaches a peak ($\sim$ 14 tps) at an MPL of 50 while for PL, the peak ($\sim$ 6 tps) is reached at an MPL of 60. Beyond that point, the throughput starts decreasing as the system starts thrashing. So GL achieves significantly higher throughput compared to PL for this dataset.

We also varied the system load by tweaking the other parameters like write probability, transaction size and size of search [1]. All these experiments were conducted on the point dataset.

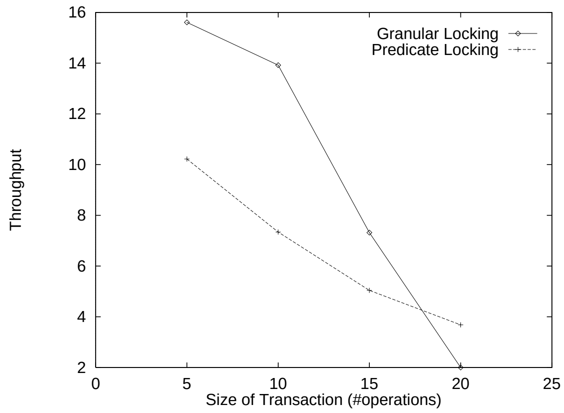

Figure 4.3: Throughput at various transaction sizes for 2-d point data (MPL=50, write probability=0.1, query selectivity=0.1%).

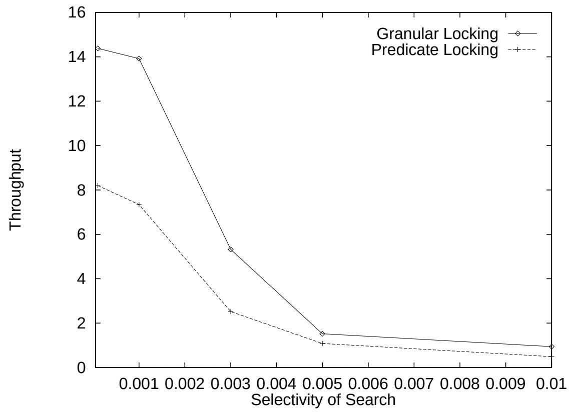

Figure 4.4: Throughput at various query sizes for 2-d point data (MPL=50, transaction size=10, write probability=0.1).

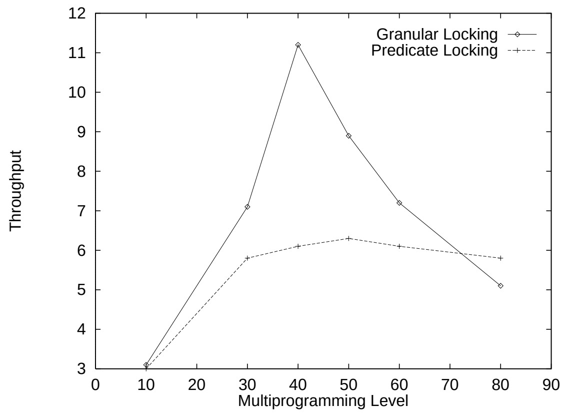

Figure 4.5: Throughput at various MPLs for 2-d polygon data (write probability=0.2, transaction size=10, query selectivity=0.1%).

Figure 4.2 shows the performance of the two protocols under various mixes of read(search) and write(insert) operations. GL significantly outperforms PL under all workloads. Figure 4.3 shows the throughputs at various transaction sizes. Again, GL mostly outperforms PurePL. At an MPL of 50, for transactions with 20 or more operations, a large portion of the R-tree is locked by some transaction or the other. As a result, GL starts thrashing due to high lock contention leading to decrease in throughput. Figure 4.4 shows the performance for various query sizes. Once again, GL performs better than PL for all workloads.

Figures 4.5 shows the performance of the two protocols for the polygon dataset. As in the point dataset, GL achieves significantly outperforms PL in terms of peak throughput.

From the experiments, we can conclude that although the granular locking approach provides slightly lower concurrency and has a small I/O overhead, it performs significantly better than predicate locking for spatial datasets.

4.3 Comparison to Other Techniques

There has been little research on providing concurrency control solutions for multidimensional index structures. The only other work we are aware of is the approach developed in [13] which uses a modified predicate locking mechanism to provide phantom protection over GiSTs. We refer to this approach as *PL/GiST*. In this protocol, a searcher attaches its search predicate Q to all the index nodes whose BPs are consistent with Q. Subsequently, the searcher acquires S locks on all objects consistent with Q. An inserter checks the object to be inserted against all the predicates attached to the node in which the insertion takes place. If it conflicts with any of them, the inserter also attaches its predicate to the node (to prevent starvation) and waits for the conflicting transactions to commit. If the insertion causes a BP of a node N to grow, the predicate attachments of the parent of N is checked with new BP of N and are replicated at N if necessary. The process is carried out top-down over the entire path where node BP adjustments take place. Similar predicate checking and replication is done between sibling nodes during split propagation. The details of the protocol can be found in [13]. Table 4.2 shows the comparison of the granular locking approach developed in this thesis with PL/GiST [13].

In the case of B-trees, granular locking is *always better* than predicate locking as it implements predicate locks much more efficiently without imposing any additional overhead to any of the operations. In case of R-trees, the granular locking approach offers slightly lower concurrency compared to the predicate approach and also imposes a small I/O overhead on the insertion operation. Our choice of granules ensures that the loss of concurrency is not significant. Our experiments show that the overhead is minimal (cf. Section 3.4.2) compared to the cost of maintaining predicates in predicate-based approaches. A more conclusive comparison between the performance of the two approaches is possible only through extensive experimentation under varying system loads. This would require a full fledged implementation of the PL/GiST protocol (including implementation of the Predicate Manager, augment GiST with data structures to be able to attach/detach predicates

Criterion	Predicate Locking Approach (PL/GiST)	Granular Locking Approach (GL)
Number of locks	Search operation acquires more locks since locks are acquired on all overlapping objects.	Search operation acquires significantly lower number of locks compared to PL since locks are acquired on all overlapping granules.
Concurrency	Provides the maximum permissible concurrency	Provides lower concurrency compared to PL. The loss in concurrency is minimized by choosing finest possible granules
Scalability	Less scalable to high system loads since under a mixed load of searchers and inserters, the lock overhead increases with the system load. The reason is that inserters needs to check the object to be inserted against the predicates of the concurrently executing transactions for potential conflict.	More scalable to high system loads since the lock overhead is independent of the number of concurrent transactions.
Complexity of search keys	*Overhead increases with the complexity of the key space* due to the increase in the cost of checking for predicate satisfiability for complex predicates.	Complexity of the key space poses no overhead since no predicate checking is required for the purpose of phantom protection.
Other Overhead	Space overhead of storing predicates e.g., managing list of predicates per transaction, list of node attachments per transaction, list of predicates attached to each node etc. Overhead of predicate management e.g., attaching/detaching by searchers to/from nodes of the tree, predicate checking during BR adjustments/node splits.	I/O overhead on insertions that change the granule boundaries. The amortized overhead over all insertions is quite small.

Table 4.2: Comparison of the predicate locking approach with the granular locking approach.

to tree nodes etc.). Due to the complexity of the this task, we do not attempt such an empirical study in this thesis and restrict ourselves to a qualitative comparison between the approaches.

Chapter 5

Conclusions

Modern database systems are commonly used to store spatial data. Spatial data is usually accessed based on its spatial location. Traditional database index structures are not suitable for spatial indexing since (1) the key space is multidimensional and (2) support for range search is required. To meet the needs of spatial indexing, several multidimensional data structures (e.g., R-trees [9], grid files [20], hB-trees [5]) have been proposed over the last decade. Despite extensive research in multidimensional indexing, none of the multidimensional data structures have been integrated as an access method to any industrial strength database system. One of the primary reasons is the lack of concurrency control protocols that guarantee consistency in the presence of concurrent operations. Permitting concurrent access to data via multidimensional index structures introduces two independent concurrency control problems. Firstly, techniques must be developed to ensure the consistency of the data structure in presence of concurrent insertions, deletions and updates. Secondly, mechanisms to protect search regions from phantom insertions and deletions must be developed. Developing such mechanisms to guarantee transactional access to data via multidimensional index structures has been identified as one of the key challenges to transaction management in future database systems [6].

This thesis addresses the problem of phantom protection in multidimensional access methods. This problem has been previously studied in [13] which proposes a predicate locking mechanism to solve the phantom problem in GiSTs. In contrast, we present a granular locking approach to phantom protection in multidimensional access methods. The granular locking approach may offer slightly lower concurrency compared to predicate locking but it is preferred to the latter as granular locks can be implemented more efficiently compared to predicate locks. Our experiments confirm that similar to granule based protocols for 1-d datasets, the granular locking protocol performs significantly better than predicate locking for multidimensional datasets as well. We present the solutions for R-trees, one of the most popular spatial index structures. To the best of our knowledge, this thesis provides the first solution to the phantom problem in multidimensional data structures based on granular locking.

The work in this thesis can be extended in several directions. One task is to perform an extensive experimental study comparing the granular locking approach developed in this thesis to the predicate locking approach proposed in [13] in terms of performance. Another important direction of research is to generalize the granular locking approach presented in this thesis so that it can applied to any multidimensional index structure rather than just R-trees. Since several multidimensional index structures have been proposed in the last few years, a general solution would allow any or all of them to be integrated to a DBMS as access methods without requiring to develop concurrency control solutions for each individual index structure. Another avenue of research is to integrate the granular locking approach developed in this thesis with techniques to ensure tree consistency. Such an integrated approach will form the complete solution to the concurrency control problem in multidimensional access methods.

Appendix A

OSCI and CSOI Policies

A.1 OSCI (Overlap-for-Search and Cover-for-Insert) Policy

Lemma 1 (Correctness of OSCI Policy) *The OSCI policy guarantees phantom protection assuming the operations do not change the granules.*

Proof: Let $Ins(O)$ be an insertion operation inserting object O and $Scan(R)$ be a search operation scanning region R. Let $lset(\mathcal{O})$ denote the set of locks acquired by an operation $\mathcal{O}$. We need to prove

$$O \cap R \Rightarrow (lset(Ins(O)) \cap' lset(Scan(R)) \tag{A.1}$$

where $\cap$ and $\cap'$ denote geometric and set intersections respectively. $R1 \cap R2$ is true if regions $R1$ and $R2$ intersect geometrically. $S1 \cap' S2$ is true if the intersection of sets $S1$ and $S2$ is not NULL. The same holds for the union ($\cup$) operation.

In OSCI policy,

$$lset(Ins(O)) = \{g_i, i = [1, n]\} \text{ such that } O \Rightarrow \bigcup_i g_i \tag{A.2}$$

and

$$h \cap R \Rightarrow h \in lset(Scan(R)) \tag{A.3}$$

Let us assume $O \cap R$ is true. Since $O \Rightarrow \bigcup_i g_i$,

$$O \cap R \Rightarrow (\bigcup_i g_i) \cap R \tag{A.4}$$

$$\Rightarrow \bigvee_i (g_i \cap R) \tag{A.5}$$

$$\Rightarrow \bigvee_i (g_i \in lset(Scan(R))) \tag{A.6}$$

$$\Rightarrow (\bigcup_i^{\prime} g_i) \cap' lset(Scan(R)) \tag{A.7}$$

$$\Rightarrow lset(Ins(O)) \cap' lset(Scan(R)) \tag{A.8}$$

■

A.2 CSOI (Cover-for-Search and Overlap-for-Insert) Policy

Lemma 2 (Correctness of CSOI Policy) *The CSOI policy guarantees phantom protection assuming the operations do not change the granules.*

Proof: Similar to above. ■

Appendix B

Lock Protocols

In the appendix, we present the lock protocols for each of the operations. The description of the algorithms closely follow the ones in [9]. The calls for acquiring and releasing the locks have been added. For completeness, all the algorithms are included.

B.1 Search

Algorithm Search(R, q, t)	
Input:	R-tree rooted at R, predicate q, transaction t
S1:	If R is a non-leaf node, check each entry E on R to determine whether q overlaps with $E.I$. If q overlaps with exactly one entry, say F, and $q \Rightarrow F.I$ (i.e. $F.I$ fully covers q), go to S2. Else acquire an S mode (commit duration) lock on $ext(R)$.
S2:	For each entry E overlapping with q, invoke **Search** on the node pointed by $E.ptr$ with $(q \cap E.I)$ as the search predicate.
S3:	If R is a leaf, acquire an S mode (commit duration) lock on $TG(R)$. Check each entry E on R to determine whether q overlaps with $E.I$. If so, it is a qualifying entry that can be returned to the calling process.

Table B.1: Search algorithm.

B.2 Insertion Algorithm

Algorithm Insert(R, E, t)	
Input:	R-tree rooted at R, entry E=(I, tuple-identifier) to be inserted, transaction t.
I1:	Invoke **ChooseLeaf** to select a leaf node L in which to place E.
I2:	If $E.I \not\Rightarrow BR(L)$ i.e. $E.I$ is not fully covered by $BR(L)$, let GR represent the growth of L. $GR = BR(L \cup E) - BR(L)$. Acquire short duration IX locks on all leaf granules overlapping with GR (released in Step AT4 of AdjustTree). Let M be the set of locks acquired. If $E.I \Rightarrow BR(L)$, M is set to NULL.
I3:	If L has room for another entry, acquire an IX mode (commit duration) lock on $TG(L)$. Install E in L. Otherwise acquire a short duration SIX lock on $TG(L)$. Invoke **SplitNode** to obtain L and LL containing E and all the old entries of L. Acquire an IX mode (commit duration) lock on either $TG(L)$ or $TG(LL)$, whichever contains E. Acquire an X lock on E.
I4:	Invoke **AdjustTree** on L. LL is also passed if the split was performed. Otherwise, LL is set to NULL. M, the set of locks acquired in Step $I2$ (NULL is none acquired), is also passed.
I5:	If node split propagation caused the root to split, create a new root whose children are the two resulting nodes. If old root is a non-leaf node and t acquired SIX lock on $ext(old_root)$ (in Step AT3 of AdjustTree), release that lock.

Table B.2: Insert algorithm.

Algorithm ChooseLeaf(R, E, t)	
Input:	R-tree rooted at R, entry E=(I, tuple-identifier) to be inserted, transaction t.
CL1:	Set N to the the root node R.
CL2:	If N is a leaf, return N.
CL3	If N is not a leaf, let F be the entry in N whose rectangle $F.I$ needs the least enlargement to include $E.I$. Resolve ties by choosing the entry with the rectangle of smallest areas.
CL4:	Set N to the child node pointed by $F.p$ and repeat from CL2.

Table B.3: ChooseLeaf algorithm.

Algorithm AdjustTree(R, L, LL, M, t)	
Input:	R-tree rooted at R, leaf node L in which insertion took place, leaf node LL if L was split (LL is NULL if L was not split), set of locks M, transaction t.
AT1:	Set $N = L$ and $NN = LL$.
AT2:	If N is the root, stop.
AT3:	Let P be the parent of N, and let E_N be N's entry in P. If either $NN \neq NULL$ OR $BR(N) \not\Rightarrow E_N.I$, acquire a short duration SIX lock on $ext(P)$ (released in Step AT4 of next iteration). Adjust $E_N.I$ so that it tightly encloses all entry rectangles in N.
AT4:	If N is a leaf node and $M \neq NULL$, release all locks in M (was acquired in Step I2 of Insert). If N is a leaf node and $NN \neq NULL$, release SIX lock on $TG(N)$ (was acquired in Step I3 of Insert). If N is a non-leaf node and t acquired SIX lock on $ext(N)$ (in Step AT3 of previous iteration), release that lock.
AT4:	If $NN \neq NULL$, create a new entry E_{NN} with $E_{NN}.p$ pointing to NN and $E_{NN}.I$ enclosing all rectangles in NN. Add E_{NN} to P if there is room. Otherwise invoke **SplitNode** to produce P and PP containing E_{NN} and all P's old entries.
AT5:	Set $N = P$. If split occurred, set $NN = PP$; else set $NN = NULL$. Repeat from AT2.

Table B.4: AdjustTree algorithm.

B.3 Deletion Algorithm

Algorithm LogicalDelete(R, E, t)	
Input:	R-tree rooted at R, entry E=(I, tuple-identifier) to be deleted, transaction t.
LD1:	Invoke **FindLeaf** to locate the leaf node L containing E. If record was not found, acquire S locks on all granules overlapping with $E.I$ (like a Search operation with $E.I$ as search predicate) and stop.
LD2:	Acquire an IX mode commit duration lock on $TG(L)$ and an X lock on E. Mark E as deleted.

Table B.5: Logical delete algorithm.

Algorithm FindLeaf(R, E, t)	
Input:	R-tree rooted at R, entry E=(I, tuple-identifier) to be deleted, transaction t.
FL1:	If R is not a leaf, check each entry F in R to determine if $F.I$ overlap $E.I$. For each such entry, invoke **FindLeaf** on the tree whose root is pointed by $F.p$ until E is found or all entries have been checked.
FL2:	If R is a leaf, check each entry to see if it matches E. If E is found, return R.

Table B.6: FindLeaf algorithm.

Algorithm DeferredDelete(R, E, L, t)	
Input:	R-tree rooted at R, entry E=(I, tuple-identifier) marked delete, node L containing E, transaction t.
DD1:	Remove E from L. Invoke **CondenseTree**, passing L.
DD2:	If root node has only one child after the tree has been adjusted, make the child the new root.

Table B.7: Deferred delete Algorithm.

Algorithm CondenseTree(R, L, t)	
Input:	R-tree rooted at R, leaf node L in which deletion took place, transaction t.
CT1:	Set $N = L$. Set Q, the set of eliminated nodes, to be empty.
CT2:	If N is the root, go to CT6. Otherwise, let P be the parent of N and E_N be N's entry in P.
CT3:	If N has fewer than m entries, acquire instant duration SIX lock on $TG(N)$ if N is a leaf node or on $ext(N)$ if N is a non-leaf node. Remove E_N from P and add N to set Q.
CT4:	If N has not been eliminated, if $E_N.I \not\Rightarrow BR(N)$ and N is a non-leaf node, acquire an instant duration SIX lock on $ext(N)$. Adjust $E_N.I$ to tightly contain all entries in N.
CT5:	Set $N = P$ and repeat from $CT2$.
CT6:	Reinsert all entries of nodes in set Q. For reinsertion of an object, the lock requirement is the same as in insert algorithm. For reinsertion of an higher level node N, IX mode locks are acquired on all granules overlapping with $N.I$.

Table B.8: CondenseTree algorithm.

B.4 UpdateScan Algorithm

Algorithm UpdateScan(R, q, t)	
Input:	R-tree rooted at R, scan predicate q, transaction t
US1:	If R is a non-leaf node, check each entry E on R to determine whether q overlaps with $E.I$. If q overlaps with exactly one entry E_0 and $q \Rightarrow E_0.I$ (i.e. $E_0.I$ fully covers q), go to S2. Else acquire an S mode (commit duration) lock on $ext(R)$.
US2:	For each entry E overlapping with q, invoke **UpdateScan** on the node pointed by $E.ptr$ with $q \cap E.I$ as the scan predicate.
US3:	If R is a leaf, acquire an S mode (commit duration) lock on $TG(R)$. Set IXLockFlag to FALSE. Check each entry E on R to determine whether q overlaps with $E.I$. If so, it is a qualifying entry that can be returned to the calling process.
US4:	If t wishes to update E, do the following: (1) Acquire an X lock on E (2) If IXLockFlag is FALSE, acquire an IX lock on $TG(R)$ and set IXLockFlag to TRUE. If IXLockFlag is TRUE, do nothing.

Table B.9: UpdateScan algorithm.

References

[1] R. Agrawal, M. Carey, and M. Livny. Models for studying concurrency control performance: Alternatives and implications. In *SIGMOD*, May 1985.

[2] ANSI. Ansi x3.135-1992, american national standard for information systems - database language - sql. November, 1992.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD*, May 1990.

[4] D. J. DeWitt., N. Kabra, J. Luo, J. M. Patel, and J. B. Yu. Client server paradise. In *Proceedings of VLDB*, September 1994.

[5] G. Evangelidis, D. Lomet, and B. Salzberg. The hb^{π}-tree: A modified hb-tree supporting concurrency, recovery and node consolidation. In *Proceedings of VLDB*, 1995.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[7] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of ICDE, pages 606-615*, 1989.

[8] The POSTGRES group. Postgres reference manual. In *Technical Report, Electronic Research Laboratory, University of California, Berkeley*, April 1994.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf., pp. 47–57.*, 1984.

[10] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized search trees in database systems. In *Proceeding of VLDB, pages 562-573*, September 1995.

[11] Informix Software Inc. Informix universal server reference manual. 1997.

[12] M. Kornacker and D. Banks. High-concurrency locking in r-trees. In *Proceedings of Very Large Databases (VLDB), pages 134-145*, September 1995.

[13] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency and recovery in generalized search trees. In *Proc. of SIGMOD*, 1997.

[14] D. Lomet. A review of recent work on multi-attribute access methods. In *SIGMOD Record*, Sept. 1992.

[15] D. Lomet. Key range locking strategies for improved concurrency. In *VLDB Proceedings*, August 1993.

[16] J. Melton and A. R. Simon. Understanding the new sql: A complete guide. *Morgan Kauffman*, 1993.

[17] C. Mohan. ARIES/KVL: A key value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceeding of VLDB*, August 1990.

[18] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, Vol. 17, No. 1:94–162, March 1992.

[19] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming.* O'Reilly & Associates, 1996.

[20] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems(TODS)*, 1984.

[21] J. Orenstein and T. Merett. A class of data structures for associative searching. In *Proc. Third SIGACT News SIGMOD Symposium on the Principles of Database Systems, pages 181-190*, 1984.

[22] B. Salzberg. Grid file concurrency. In *Information and Systems*, volume 11(3): 235–244, 1986.

[23] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company, Inc, 1990.

[24] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proc. VLDB*, 1987.

[25] M. Stonebraker, J. Frew, K. Gardels, and J Meredith. The sequoia 2000 storage benchmark. *Proc. of SIGMOD*, 1993.